THE LOGIC IN COMPUTER SCIENCE COLLUMN
by
Yuri GUREVICH[*]

How to find a coin: propositional program logics made easy[†]

N.V. Shilov[‡] K. Yi[⋆]

[‡]A.P. Ershov Institute of Informatics Systems
Siberian Division of Russian Academy of Science
Novosibirsk, Russia
email: `shilov@iis.nsk.su`

[⋆]School of Computer Science and Engineering
Seoul National University
Seoul, KOREA
email: `kwang@ropas.snu.ac.kr`
homepage: `ropas.snu.ac.kr/∼kwang`

1 Introduction

In spite of the importance of propositional program logics (PPLs) to the development of reliable software and hardware, the area is not well known to non-professionals. In particular, many hacker programmers and hardware designers presume that PPLs are too pure for their poor mathematics, while many mathematical purists believe that PPLs are too poor for their pure mathematics. After

[*]`http://reasearch.microsoft.com/∼gurevich`

[†]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology

25 years of progress in PPLs research a lot of hardware and software people remain illiterate in PPLs and reluctant to apply them [26].

Maybe, a deficit of popular papers on the topic is the main reason for this ignorance. Our article is aiming to overcome this deficit and presents propositional program logics in a popular (but mathematically sound) manner. The basic ideas, definitions and properties are illustrated by puzzles and game examples. In particular, the fable of the article is a model-checking-based solution for a complicated puzzle to identify a single false coin among given ones balancing them limited times. Only some knowledge of propositional calculus and elementary set theory is assumed.

The rest of the article is organized as follows. The balancing puzzle and a related programming problem are discussed informally in Section 2. Then Section 3 introduces formalisms of finite games, Elementary Propositional Dynamic Logic (EPDL) [14] and discusses utility of this logic for reasoning about finite games. Notions of model checking and abstraction are also introduced and illustrated in Section 3. The propositional μ-Calculus (μC) [17] is defined in the section 4 as an extension of EPDL. The expressive power of μC and its utility for solving finite games are discussed in the same section. A brief survey (but with some formal details) of basic algorithmic problems for the propositional μ-Calculus (i.e., model checking, decidability, and axiomatization) is presented in Section 5. In contrast, a purely informal survey of program logic history and research is given in Section 6. Finally we give in Section 7 a high-level design (in terms of μC model checking and abstraction) of an efficient algorithm for the programming version of the coin balancing puzzle which was introduced in Section 2.

We would like to conclude this introduction with many thanks to Andrzej Murawski and Mike Barnett. Their very useful remarks and language suggestions have helped us to make propositional program logics easy.

2 How the story began

2.1 A hard puzzle

Once upon a time a program committee for a regional middle school contest in mathematics discussed problems for a forthcoming competition. The committee comprised a professor, several Ph.D. holders and a couple of Ph.D. students. All were experienced participants or organizers of mathematical contests on the regional and national level; several had international experience. A thunderstorm was unexpected, but it came. Suddenly (when a set of problems was almost complete) one of the youngest participants suggested another problem to be included. It was the following 15-coin puzzle:

• A set of 15 coins consists of 14 valid coins and a false one. All valid coins have one and the same weight while the false coin has a different weight. One of the valid coins is marked but all other coins (including the false one) are unmarked. Is it possible to identify the false coin using a balance at most 3 times?

"If it is known that the false coin is heavier than the valid one then the problem is suitable for 11–13-year-old school-children" the professor said. "Sorry, but it is not known whether the false coin is heavier or lighter than the valid one," the youngster replied and added: "To the best of my knowledge it is really a hard puzzle. I did not solve it during the national contest several years ago and I still do not know a solution". "In this case it would be better not to include the puzzle into the problem list for the forthcoming contest" the professor concluded.

Problem 1 *Solve the 15-coin puzzle.*

One of the committee members was a computer scientist specializing in program logics and their applications. He agreed to exclude the puzzle from the problem list, since he could not solve it either. Nevertheless he was concerned and decided to try his luck. The day passed without real progress while the coins and balance became his nightmare... The following morning the computer scientist decided to overcome the trouble and adopted the following plan with two concurrent approaches to finding a solution:

- human-aided,
- computer-aided.

The computer-aided approach and its relation to propositional program logics is the main topic of the paper. As far as the human-oriented approach is concerned, the idea was very simple: the puzzle was offered (with a special bonus for the first solution[1]) to students and faculty of the Mathematics Department. But the first who solved the puzzle was the computer scientist's wife[2]. A week later the computer scientist got several correct (and very similar) solutions from students. Then several weeks later some faculty members solved the puzzle correctly as well...

But the human-aided approach had some unexpected implications. The University where the computer scientist is employed is situated in a cozy scientific town,[3] not in a political, industrial or financial center. Several months later a

[1]The bonus was a photocopy of $100.

[2]So the bonus remained at home.

[3]Can you guess the name of the town and where it is?

local book seller began to offer a special deal for textbooks: if a customer could find in one hour a strategy to identify a single false coin among 39 coins with aid of a marked valid coin[4] and weighing coins at most 4 times, then the customer would get his money back; if the customer could find a strategy in one day then he would get 50% of his money back.

2.2 Put it for programming

Let us turn to the computer-aided approach and understand how the 15-coin puzzle can be be generalized for programming. The problem is not to identify a false coin but to find *a strategy* to identify this coin. From the programming viewpoint, a natural model for a strategy is a program which chooses the next step using the information available after previous steps. In this setting the following programming problem is a natural generalization and formalization of the 15 coins puzzle:

- Write a program with 3 inputs

  - a number $N \geq 0$ of coins under question,
  - a number $M \geq 0$ of marked valid coins,
  - a limit $K \geq 0$ of the number of weighings

  which outputs either the string `impossible`, or another executable interactive program `ALPHA` (in the same language) with respect to existence of a strategy to identify a single false coin among N coins with use of additional M marked valid coins and weighing coins K times at most. Your program should output `impossible` iff there is no such strategy. Otherwise it should output the program `ALPHA` which implements a strategy in the following settings.

  All $(N + M)$ coins are enumerated by consecutive numbers from 1 to $(N + M)$, all marked valid coins are enumerated by initial numbers from 1 up to M. These are called *coin numbers*.

  Every interactive session with `ALPHA` begins with user's initial decision on the coin number of the false coin in $[(M+1)..(N+M)]$ and whether it is lighter or heavier.

  Every interactive session with `ALPHA` consists of a series of rounds and the number of rounds in the session can not exceed K. In each round i $(1 \leq i \leq K)$ the program `ALPHA` outputs two disjoint subsets of coin

[4]Valid coins have the same weight while the false coin has a different weight.

```
user:  2nd is heavier  3rd is lighter  4th is heavier  5th is lighter
prog:  {1,2} {3,4}     {1,2} {3,4}     {1,2} {3,4}     {1,2} {3,4}
user:       >               >               <               =
prog:    {3} {4}         {3} {4}         {3} {4}         {1} {5}
user:       =               <               <               >
prog:       2               3               4               5
```

Figure 1: A summary of four sessions with this program `ALPHA` (Fig. 2) which identifies a single false coin among five coins with aid of a special marked valid coin and weighing coins at most twice.

numbers to be placed on the left and the right pans of the balance and prompts the user with `?` for a reply. The user in his/her turn replies with $<$, $=$ or $>$ in accordance with the initial decision on the number of the false coin and its weight.

Every interactive session with `ALPHA` finishes with the final output string `false coin number is` followed by the coin number of the false coin.

Since the problem is to write a program which generates another program we would like to refer to the first program as the *metaprogram* and to the problem as the *metaprogram problem* respectively. For the first time the problem was designed and offered for training university students for a regional edition of the 1999 ACM Collegiate Programming Contest [33].

Problem 2 *Solve the metaprogram problem.*

Let us illustrate the metaprogram problem by examples of inputs/outputs.

The triple 5, 1 and 2 is an example of possible input for a metaprogram. The semantics of this particular input is a request for a strategy which can identify a single false coin among five coins ($N = 5$) using an additional marked valid coin ($M = 1$) and balancing coins at most twice ($K = 2$). All $6 = (N + M)$ coins are enumerated by consecutive numbers from 1 to 6, and the unique marked valid coin has the number 1. The PASCAL program `ALPHA` presented in Fig. 2 is a possible corresponding output of the metaprogram. A summary of four sessions with `ALPHA` is presented in Fig. 1.

The triple 9, 1 and 2 is another example of valid input. It asks for a strategy which can identify a single false coin among nine coins with the aid of an extra marked valid coin and balancing coins at most twice. The correct output of the metaprogram for this particular input is `impossible`.

```
program ALPHA
var R: '<','=','>';
begin
writeln(1,2); writeln(3,4); writeln('?'); readln(R);
if R='='
   then
     begin
     writeln(1); writeln(5); writeln('?'); readln(R);
     if R='='
        then writeln('false coin number is 6')
        else writeln('false coin number is 5')
     end
   else
     if R='<'
        then
          begin
          writeln(3); writeln(4); writeln('?'); readln(R);
          if R='='
             then writeln('false coin number is 2')
             else if R='<'
                     then writeln('false coin number is 4')
                     else writeln('false coin number is 3')
          end
        else
          begin
          writeln(3); writeln(4); writeln('?'); readln(R);
          if R='='
             then writeln('false coin number is 2')
             else if R='<'
                     then writeln('false coin number is 3')
                     else writeln('false coin number is 4')
          end
end.
```

Figure 2: A PASCAL program which identifies a single false coin among five coins with the aid of a special marked valid coin and balancing coins at most twice.

3 Games with Dynamic Logic

3.1 Game interpretation

The examples of sessions in Fig. 1 naturally lead to a game interpretation for 15-coin puzzle and metaprogram problem.

> • Let M and N be non-negative integer parameters and let $(N+M)$ coins be enumerated with consecutive numbers from 1 to $(N+M)$. Coins with numbers in $[1..M]$ are valid while there is a single false coin among those with numbers in $[(M+1)..(M+N)]$. The *GAME(N,M)* between two players *user* and *prog* consists of a series of rounds. In each round a move of *prog* is a pair of disjoint subsets (with equal cardinalities) of $[1..(M+N)]$. A possible move of *user* is either $<$, $=$ or $>$, but the reply should be *consistent* with all previous rounds in the following sense: some number in $[1..(M+N)]$ and the weight of the false coin meet all constraints induced on the current and previous rounds. *Prog* wins the *GAME(N,M)* as soon as a *unique* number in $[1..(M+N)]$ satisfies all constraints induced during the game.

Now problems 1 and 2 can be reformulated as follows.

1. Find a 3-round (at most) winning strategy for *prog* in the $GAME(14,1)$.
2. Write a metaprogram which for any $N \geq 1$, $K \geq 0$ and $M \geq 0$ generates (if possible) a K-round (at most) winning strategy for *prog* in the *GAME(N,M)*.

The above game interpretation is still too complicated for analysis. So let us introduce and analyze another simpler game example, namely, the following Millennium Game Puzzle:

> •On the eve of New Year 2001 Alice and Bob play the *millennium game*. Positions in the game are dates in 2000 and 2001. An *initial position* is a random date of the year 2000. Then Alice and Bob made alternating moves: Alice, Bob, Alice, Bob, etc. Available moves are one and the same for both Alice and Bob: if a current position is a *date* then *the next calendar date* or *the same day of the next month* are possible next positions. A player wins the game iff the other player is the first to launch the year 2001. Problem: Find all initial positions where Alice has a winning strategy.

A mathematical model for the millennium game is quite obvious. It is a directed labeled graph $G_{2000/2001}$. Nodes of this graph correspond to game positions: dates of years 2000 and 2001. All dates of the year 2001 are marked by *fail* while all other dates are unmarked. Edges of the graph correspond to possible moves and are marked by *move*. We would like to consider *fail* and *move* as variables for collections of nodes and sets of edges and call them propositional and action variables respectively. The model fixes the interpretation (i.e. values) of these variables in the manner described above.

In general, a *finite game of two players* A and B is a tuple $(P,\ M_A,\ M_B,\ F)$ where

- P is a nonempty finite set of *positions*,
- $M_A, M_B \subseteq P \times P$ are *moves* of A and B,
- $F \subseteq P$ is a set of *final positions*.

A *session* (or *play*) of the game is a finite sequence of positions $s_0,\ ...,\ s_n$ $(n > 0)$ where all even pairs are moves of one player (ex., all $(s_{2i},\ s_{2i+1})\ \in M_A$) while all odd pairs are moves of another player (ex., all $(s_{2i+1},\ s_{2i+2})\ \in M_B$). A pair of consecutive moves of two players in a session comprising three consecutive positions (ex., $(s_{2i},s_{2i+1},s_{2(i+1)})$ is called a *round*. We say that a session $s_0,\ ...,\ s_n$ consists of $\left[\frac{n+1}{2}\right]$ rounds. A player *loses* a session iff after a move of the player the session enters a final position for the first time. A player *wins* a session iff the other player loses the session. A *strategy* of a player is a subset of the player's possible moves. A *winning strategy* for a player is a strategy of the player which always leads to the player's win: the player wins every session which he/she begins and in which he/she implements this strategy instead of all possible moves. The millennium game is just an example of a finite game.

Finite games of two players can easily be presented as directed labeled graphs. Nodes correspond to game positions, those which correspond to final positions are marked by $fail$, while all other nodes are unmarked. Edges of these graphs correspond to possible moves of players and are marked by $move_A$ and $move_B$ respectively. Let us denote by $G_{(P,M_A,M_B,F)}$ the labeled graph corresponding to a game (P, M_A, M_B, F). We would like to consider $fail$, $move_A$ and $move_B$ as variables for sets of nodes and sets of edges respectively.

3.2 Elementary Propositional Dynamic Logic

Let $\{true, false\}$ be boolean constants, *Prp* and *Act* be disjoint finite alphabets of *propositional* and *action* variable respectively. (In the previous subsection they are $\{fail\}$ and $\{move,\ move_A,\ move_B\}$.)

The syntax of the classical propositional logic consists of *formulae* which are constructed from propositional variables and boolean connectives $\neg$ (*negation*), $\wedge$ (*conjunction*) and $\vee$ (*disjunction*) in accordance with the standard rules:

1. all propositional variables and boolean constants are formulae;
2. if ϕ is a formula then $(\neg\phi)$ is a formula;
3. if ϕ and ψ are formulae then $(\phi \wedge \psi)$ is a formula,
4. if ϕ and ψ are formulae then $(\phi \vee \psi)$ a formula.

Elementary Propositional Dynamic Logic (EPDL)[14] has additional formula constructors, modalities, which are associated with action variables:

5. if a is an action variable and ϕ is a formula then $([a]\phi)$ is a formula[5],
6. if a is an action variable and ϕ is a formula then $(\langle a\rangle\phi)$ is a formula[6].

We would like to use several standard abbreviations $\rightarrow$ and $\leftrightarrow$ with the usual meaning: if ϕ and ψ are formulae then $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are abbreviations for formulae $((\neg\phi) \vee \psi)$ and $((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$ respectively. Then we would like to avoid extra parentheses and use the standard precedence of connectives and modalities: $\neg$, $\langle\rangle$, $[\ ]$, $\wedge$, $\vee$, $\rightarrow, \leftrightarrow$. We also use the *meta-symbol* $\equiv$ for *syntactical equality.*

The semantics of EPDL is defined in models, which are called *labeled transition systems* by computer scientists and *Kripke structures* by mathematicians. A model M is a pair (D_M, I_M) where the *domain* D_M is a nonempty set, while the *interpretation* I_M is a pair of special mappings (P_M, R_M). Elements of the domain D_M are called *states.* The interpretation maps propositional variables to sets of states and action variables to binary relations on states:

$$P_M : Prp \rightarrow \mathcal{P}(D_M) \quad , \quad R_M : Act \rightarrow \mathcal{P}(D_M \times D_M)$$

where $\mathcal{P}$ is the power-set operation. We write $I_M(p)$ and $I_M(a)$ instead of $P_M(p)$ and $R_M(a)$ frequently whenever it is implicit that p and a are propositional and action variables respectively.

Every model $M = (D_M, I_M)$ can be viewed as a directed graph with nodes and edges labeled by sets of propositional and action variables respectively. Its nodes are states of D_M and a node $s \in D_M$ is marked by a propositional variable $p \in Prp$ iff $s \in I_M(p)$. A pair of nodes $(s_1, s_2) \in D_M \times D_M$ is an edge of the

[5]which is read as "*box a* ϕ" or "*after a always* ϕ"

[6]which is read as "*diamond a* ϕ" or "*after a sometimes* ϕ"

graph iff $(s_1, s_2) \in I_M(a)$ for some action variable $a \in Act$; the edge (s_1, s_2) is then marked with the action variable a. Conversely, graphs with nodes and edges labeled by sets of propositional and action variables respectively can be considered as models too. Thus the graph $G_{2000/2001}$ of the millennium game is really a model for EPDL as well as the graph $G_{(P,M_A,M_B,F)}$ of a game (P, M_A, M_B, F).

For every model $M = (D_M, I_M)$ the *validity* relation $\models_M$ between states and formulae can be defined inductively with respect to the structure of formulae:

1. for every state $s \models_M true$ and not $s \models_M false$;
   for all states s and propositional variables p: $s \models_M p$ iff $s \in I_M(p)$;
2. for any state s and formula ϕ:
   $s \models_M (\neg\phi)$ iff it is not the case that $s \models_M \phi$;
3. for any state s, formulae ϕ and ψ: $s \models_M (\phi \wedge \psi)$ iff $s \models_M \phi$ and $s \models_M \psi$;
4. for any state s, formulae ϕ and ψ: $s \models_M (\phi \vee \psi)$ iff $s \models_M \phi$ or $s \models_M \psi$;
5. for any state s, action variable a and formulae ϕ:
   $s \models_M (\langle a\rangle\phi)$ iff $(s, s') \in I_M(a)$ and $s' \models_M \phi$ for some state s' ;
6. for any state s, action variable a and formulae ϕ:
   $s \models_M ([a]\phi)$ iff $(s, s') \in I_M(a)$ implies $s' \models_M \phi$ for every state s'.

3.3 Finite games in EPDL

First let us illustrate the above definition by several examples in the model $G_{2000/2001}$. The formula $fail$ is valid in the states where the game is lost. Then the formula $[move]fail$ is valid in the states from which all possible moves lead to a loss. Hence the formula $\neg fail \wedge [move]fail$ is valid in the states where the game is not over but all possible moves lead to a lost game. Consequently, the formula $\langle move\rangle(\neg fail \wedge [move]fail)$ is valid iff there is a move after which the game is not lost but all possible moves lead to a lost game. Finally we get: the formula

$$\neg fail \wedge \langle move\rangle(\neg fail \wedge [move]fail)$$

is valid in those states where the game is not over, there exists a move after which the game is not lost, and then all possible moves always lead to a loss in the game. So the last EPDL formula is valid in those states of $G_{2000/2001}$ (i.e. dates of years 2000 and 2001) where Alice has a 1-round winning strategy against Bob[7]. So it

[7]Alice has all odd-numbered moves while Bob has all even-numbered moves.

is natural to denote this formula by win_1. It becomes quite clear from the above arguments that the following formula

$$\neg fail \wedge \langle move\rangle(\neg fail \wedge [move](fail \vee win_1))$$

is valid in those states of $G_{2000/2001}$ where Alice has a winning strategy with at most 2 rounds. So it is natural to denote this formula by win_2. Let us define formulae win_i for all $i \geq 1$ similarly to win_1 and win_2: for every $i \geq 1$ let

$$win_{i+1} \;\equiv\; \neg fail \wedge \langle move\rangle(\neg fail \wedge [move](fail \vee win_i)).$$

Let win_0 be $false$ in addition. After the above discussion about win_1 and win_2 it becomes quite simple to prove the following by induction:

Assertion 1 *For every $i \geq 1$ the formula win_i is valid in those states of $G_{2000/2001}$ where Alice has a winning strategy against Bob with at most i rounds.*

The following proposition is just a generalization of the above assertion 1.

Proposition 1 *Let (P, M_A, M_B, F) be a finite game of two players, a formula, WIN_0, be $false$ and for every $i \geq 1$ let WIN_{i+1} be a formula*

$$\neg fail \wedge \langle move_A\rangle(\neg fail \wedge [move_B](fail \vee WIN_i)).$$

For every $i \geq 0$ the formula WIN_i is valid in those states of $G_{(P,M_A,M_B,F)}$ where a player A has a winning strategy against a counterpart B with at most i rounds.

3.4 Model checking and abstraction

Model checking is testing a model against a formula. The *global checking* problem consists in calculation of *the set of all states* of an input model where an input formula is valid. The *local checking* problem consists in testing the boolean value of an input formula in an input state of an input model. Thus the corresponding model checking algorithms as well as their implementations (called *model checkers*) can be characterized by their inputs and outputs as shown in Fig. 3.

We are especially interested in model checking problems for finite models, i.e. models with finite domains. For these models both model checking problems are algorithmically equivalent:

- for global checking just check locally all states and then collect states where a formula is valid,
- for local checking just check globally and check whether a state is in the validity set of a formula.

	inputs	**outputs**
global	a model and a formula	all states of the model where the formula is valid
local	a model, a formula, and a state	a boolean value of the formula in the state of the model

Figure 3: Global vs. local model checking

Of course, the above reduction of global checking to local checking leads to changes in time complexity: the global checking complexity is less than or equal to the local checking complexity multiplied by the number of states. We would like to concentrate on global model checking only since this complexity difference is not important for logics discussed in the paper. A more important topic is the parameters used for measuring this complexity. If $M = (D_M, (R_M, P_M))$ is a finite model, then let d_M and r_M be the number of states in D_M and edges in R_M respectively; let m_D be the overall complexity $(d_M + r_M)$. (If the model M is implicit then we use these parameters without subscripts.) If ϕ is a formula then let f_ϕ be the size of the formula presented as a string. (If the formula ϕ is implicit then we simply write this parameter without subscript.) The following proposition is a straightforward implication of EPDL semantics.

Proposition 2 *The model checking problem for EPDL formulae in finite models is decidable with time complexity $O(m \times f)$.*

Thus the model checking complexity for EPDL in finite models is linear in both arguments: model and formula size. This upper bound seems to be pretty good and the best possible. But it is not the case due to a disproportion between the model and formula sizes which occurs frequently: models of software and hardware are very big and even huge, while logical specifications presented by formulae are comparatively small.

Let us consider the millennium game. If we would like to check in what initial dates of 2000 and 2001 Alice has a n-round winning strategy against Bob (n is a parameter) then we can model-check the EPDL formula win_n in the model $G_{2000/2001}$. This model consists of $d_{2000/2001} = 730$ positions and $r_{2000/2001} = 1415$ moves, i.e., its overall size is $m_{2000/2001} = 2145$. The size of the formula win_n is $f_n = (14 \times n - 3)$. Intuitively it seems very likely that Alice has a winning strategy against Bob iff there is a winning strategy with at most 12 rounds. But the size of the formula win_{12} is more than 12 times smaller than the size of the model $G_{2000/2001}$. In real life examples the difference between the model and formula sizes is much more serious.

There are several techniques for curbing the model size, but we would like to discuss *abstraction* only. In general, let Φ be a set of formulae, $M_1 = (I_1, D_1)$ and $M_2 = (I_2, D_2)$ be two models, and $g : D_1 \to D_2$ be a mapping. The model M_2 is called an *abstraction* [8] of the model M_1 with respect to formulae in Φ iff for any formula $\phi \in \Phi$ and any state $s \in D_1$ the following holds: $s \models_1 \phi \Leftrightarrow g(s) \models_2 \phi$.

In particular, let G_{2000} be a model where states are dates of the year 2000 extended by a special state 2001, the propositional variable $fail$ is interpreted as a singleton $\{2001\}$, and the action variable *move* is interpreted as a move from a date in 2000 to

$$\begin{cases} \text{the next calendar date, if it is still in the year 2000,} \\ \text{the same date of the next month, if it remains in 2000,} \\ \text{the special state 2001, otherwise.} \end{cases}$$

Let $abs_{2001} : D_{2000/2001} \to D_{2000}$ be the following mapping

$$\lambda\ date\ . \begin{cases} date, \text{ if } date \in \text{ year 2000} \\ 2001, \text{ if } date \in \text{ year 2001} \end{cases}$$

Assertion 2 *G_{2000} is an abstraction of $G_{2000/2001}$ with respect to EPDL formulae constructed from a single propositional variable $fail$ and a single action variable move. The corresponding abstraction mapping is abs_{2001}.*

The abstract model G_{2000} consists of $d_{2000} = 366$ states and $r_{2000} = 722$ moves, i.e., its overall size is $m_{2000} = 1088$. Thus it is almost twice as small as the original model $G_{2000/2001}$. But the abstract model preserves all EPDL properties of the original one. Hence it is possible to model-check EPDL formulae in the abstract model instead of the original. This model checking is twice as efficient due to smaller model size.

4 Propositional μ-Calculus

4.1 Toward stronger logic

We have assumed that Alice has a winning strategy against Bob in the millennium game iff there is a winning strategy with 12 rounds (at most). The only intuition behind this conjecture was: the number of months in a year is 12. But we have not proved that this 12-round hypothesis really holds. It is the first disadvantage of the conjecture.

[8] g is called an *abstraction mapping* in this case.

$$NEG\ :\ ...\xrightarrow{move}(-i-1)\xrightarrow{move}\underbrace{(-i)\xrightarrow{move}...(-1)\xrightarrow{move}\overset{fail}{(0)}}_{NEG_i}$$

Figure 4: Model NEG

The size of formulae win_1, ... win_{12} is another disadvantage of the 12-round hypothesis. Really, the formula win_3 unfolds as

$$\neg f \wedge \langle m\rangle\{\neg f \wedge [m]\Big\{f \vee \Big\{\neg f \wedge \langle m\rangle\Big(\neg f \wedge [m]\Big(f \vee (\neg f \wedge \langle m\rangle(\neg f \wedge [m]f))\Big)\Big)\Big\}\Big\}\}$$

where f and m are abbreviations for $fail$ and $move$. Formula win_{12} is, approximately, 4 times larger than win_3, thus 4 lines are necessary for its presentation.

Finally, it is not clear whether in general it is possible to express the existence of winning strategies in finite games in terms of EPDL. Informally speaking, the existence of winning strategies can be expressed by an *infinite* disjunction

$$WIN_0\ \vee\ WIN_1\ \vee\ WIN_2\ \vee\ WIN_3\ \vee\ WIN_4\ \vee\ ...\ = \bigvee_{i\geq 0} WIN_i$$

but this expression *is not a legal* formula in EPDL. The following argument proves formally that EPDL is too weak for expressing it.

Let us consider all non-positive integers as a domain and interpret $fail$ to be valid on 0 only, $move$, $move_A$, and $move_B$ to be interpreted as the successor function $\lambda x.(x+1)$ on negatives. Let us denote this model by NEG (Fig. 4). Let us define an *action nesting* for EPDL formulae by induction:

1. $nest(fail)\ =\ nest(true)\ =\ nest(false)\ =\ 0,$
2. $nest(\neg\phi)\ =\ nest(\phi),$
3. $nest(\phi \wedge \psi)\ =\ \max\{nest(\phi), nest(\psi)\},$
4. $nest(\phi \vee \psi)\ =\ \max\{nest(\phi), nest(\psi)\},$
5. $nest([move]\phi)\ =\ nest([move_A]\phi)\ =\ nest([move_B]\phi)\ =\ 1 + nest(\phi),$
6. $nest(\langle move\rangle\phi)\ =\ nest(\langle move_A\rangle\phi)\ =\ nest(\langle move_B\rangle\phi)\ =\ 1 + nest(\phi).$

In this setting, for every EPDL formula ϕ, for all $k, l > nest(\phi)$ the following can be trivially proved by induction on formula's structure:

$$(-k) \models_{NEG} \phi\ \Leftrightarrow\ (-l) \models_{NEG} \phi.$$

Thus for every formula of EPDL there exists a non-positive number prior to which the formula is a boolean constant. But the infinite disjunction $\bigvee_{i\geq 0} WIN_i$ is valid in all even negative integers, and is invalid in 0 and all odd negative integers. Finally we can remark that no EPDL formula ϕ can distinguish the finite model NEG_i (Fig. 4) with $i > nest(\phi)$ from the infinite model NEG. But every NEG_i $(i \geq 0)$ is a finite game. Thus we have proved

Assertion 3 *No EPDL formula can express the existence of winning strategies in all finite games NEG_i, where $i \geq 0$.*

So it seems reasonable to have another logic with stronger expressive power. Below we are going to describe the so-called *propositional μ-Calculus* (μC) [17] which is an extension of the EPDL. Both syntax and semantics of this logic are more complicated than those of EPDL.

4.2 μ-Calculus syntax

Let us extend the syntax of EPDL with two new features:

7. if p is a propositional variable and ϕ is a formula then $(\mu p.\phi)$ is a formula[9],

8. if p is a propositional variable and ϕ is a formula then $(\nu p.\phi)$ is a formula[10].

Informally speaking $\mu p.\phi$ and $\nu p.\phi$ are "abbreviations" for infinite disjunction and conjunction

$$false \lor \phi_p(false) \lor \phi_p(\phi_p(false)) \lor \phi_p(\phi_p(\phi_p(false))) \lor \ldots = \bigvee_{i\geq 0} \phi_p^i(false)$$

$$true \land \phi_p(true) \land \phi_p(\phi_p(true)) \land \phi_p(\phi_p(\phi_p(true))) \land \ldots = \bigwedge_{i\geq 0} \phi^i(true),$$

where $\phi_p(\psi)$ is the result of substituting ψ for p in ϕ, $\phi_p^0(\psi)$ is ψ and $\phi_p^{i+1}(\psi)$ is $\phi_p(\phi_p^i(\psi))$ for every $i \geq 0$. In particular, if ϕ is a formula

$$\neg fail \land \langle move_A\rangle(\neg fail \land [move_B](fail \lor win))$$

where win is a new propositional variable, then WIN_0 is just $\phi_{win}^0(false)$, the formula WIN_1 is equivalent to

$$\phi_{win}^1(false) \equiv \neg fail \land \langle move_A\rangle(\neg fail \land [move_B](fail \lor false))$$

[9]which is read as "*mu p ϕ*" or "*the least fixpoint p of ϕ*"

[10]which is read as "*nu p ϕ*" or "*the greatest fixpoint p of ϕ*"

and for every $i \geq 0$ formula WIN_{i+1} is equivalent to

$$\phi^{i+1}_{win}(false) \;\equiv\; \neg fail \wedge \langle move_A \rangle(\neg fail \wedge [move_B](fail \vee \phi^{i}_{win}(false))).$$

Finally, the infinite disjunction $\bigvee_{i \geq 0} WIN_i$ should be equivalent to

$$\mu\ win.\ \phi \;\equiv\; \mu\ win.\Big(\neg fail \wedge \langle move_A \rangle(\neg fail \wedge [move_B](fail \vee win))\Big).$$

Let us denote the last formula by WIN.

The above definition of formulae is too loose. We would like to impose a context-sensitive restriction. In formulae $(\mu p.\phi)$ and $(\nu p.\phi)$ the range of μp and νp is the formula ϕ and all instances of the variable p are called *bound* in $(\mu p.\phi)$ and $(\nu p.\phi)$. An instance of a variable in a formula is called *free* iff it is not bound. In a formula $(\neg\phi)$ the range of negation is the formula ϕ. An instance of a propositional variable in a formula is called *positive/negative* iff it is located in the scope of an even/odd number of negations. The context-sensitive restriction reads as follows: *No bound instance of a propositional variable can be negative.*

Thus the definition of the syntax of the μ-Calculus formulae is over. But, as usual, we would like to avoid extra parentheses and extend the standard precedence to: $\neg,\ \langle\rangle,\ [\,],\ \mu,\ ,\nu,\ \wedge,\ \vee,\ \rightarrow, \leftrightarrow$.

4.3 μ-Calculus semantics

The semantics of μ-Calculus is defined in the same models as that of EPDL in terms of sets of states in which formulae are valid. For every model $M = (D_M, I_M)$ let us denote by $M(formula)$ a set of all states of the model where the $formula$ is valid. The first 6 clauses of the definition deal with EPDL features:

1. for boolean constants $M(true) \;=\; D_M$ and $M(false) \;=\; \emptyset$;
   for every propositional variable p, $M(p) \;=\; I_M(p)$;

2. for every formula ϕ, $M(\neg\phi) \;=\; D_M \setminus M(\phi)$;

3. for all formulae ϕ and ψ, $M(\phi \wedge \psi) \;=\; M(\phi) \cap M(\psi)$;

4. for all formulae ϕ and ψ, $M(\phi \vee \psi) \;=\; M(\phi) \cup M(\psi)$;

5. for any action variable a and formula ϕ, $M(\langle a \rangle\phi) \;=$
   $\{s \in D_M : (s, s') \in I_M(a)$ and $s' \in M(\phi)$ for some state $s' \in D_M\}$;

6. for any action variable a and formula ϕ, $M([a]\phi) \;=$
   $\{s \in D_M : (s, s') \in I_M(a)$ implies $s' \in M(\phi)$ for every state $s' \in D_M\}$.

As far as new features μ and ν are concerned, let us define their semantics in *finite* models only since it is the major domain for model checking applications:

7. for every formula ϕ, $M(\mu p.\ \phi)\ =\ \bigcup_{i\geq 0} M(\phi^i_p(false))$,
8. for every formula ϕ, $M(\nu p.\ \phi)\ =\ \bigcap_{i\geq 0} M(\phi^i_p(true))$.

Let us define the *validity* relation $\models'_M$ for all formulae and states in a natural way: $s \models'_M \phi$ iff $s \in M(\phi)$. Let us remark also that we can use the same notation $\models_M$ in the framework of the μ-Calculus as in the framework of EPDL since the following holds:

Proposition 3 *The μ-Calculus is a conservative extension of EPDL: $s \models'_M \phi$ iff $s \models_M \phi$, for any EPDL formula ϕ, any model M and any state s.*

In accordance with the definition of semantics in finite models, $\mu p.\ \phi(p)$ and $\nu p.\ \phi(p)$ are really "abbreviations" for the infinite disjunction $\bigvee_{i\geq 0} \phi^i_p(false)$ and conjunction $\bigwedge_{i\geq 0} \phi^i(true)$. In particular the formula WIN is really equivalent to the infinite disjunction $\bigvee_{i\geq 0} WIN_i$. Hence, this formula of the μ-Calculus expresses the existence of winning strategies in finite games of two players. In accordance with assertion 3, it is not equivalent to any formula of EPDL. These arguments prove the following

Proposition 4 *The μ-Calculus is more expressive then EPDL. The following formula $WIN\ \equiv\ \mu\ win.\Big(\neg fail \wedge \langle move_A\rangle(\neg fail \wedge [move_B](fail \vee win))\Big)$, which expresses the existence of winning strategies in finite games, is not expressible in EPDL.*

The above formula WIN is not the only μC formula of interest. For example, let us consider another formula

$$FAIR\ \equiv\ \nu q.([a]q\ \wedge\ \mu r.(p \vee [a]r)).$$

A sub-formula $\phi\ \equiv\ \mu r.(p \vee [a]r)$ of this formula is valid in a model in the states where every infinite a-path eventually leads to p. A formula $\nu q.([a]q \wedge x)$ is valid in a model in the states where every a-path always leads to x (x is a propositional variable). Hence, $FAIR\ \equiv\ \nu q.([a]q \wedge \phi) \equiv \nu q.([a]q\ \wedge\ \mu r.(p \vee [a]r))$ is valid in a state of a model iff every infinite a-path infinitely often visits states where p holds. An infinite sequence is said to be *fair* with respect to a property iff the property holds for an infinite amount of elements of the sequence. In these terms $FAIR$ holds in a state of a model iff every infinite a-path is fair with respect to p. For example, a *scheduler* of CPU time among several permanent resident jobs job_1,... job_n is fair with respect to a concrete job_i iff it schedules this job for execution by CPU infinitely often. This fairness can be expressed by the following instance of the formula $FAIR$: $\nu q.([scheduler]q\ \wedge\ \mu r.(active(job_i)\ \vee\ [scheduler]r))$.

4.4 Properties of μC semantics

The semantics of formulae as well as the semantics of propositional variables are sets of states. This gives us a new opportunity to consider the semantics of μC formulae as functions which map interpretations of a propositional variables into sets where the formulae are valid in corresponding interpretations. For example, let ϕ be a μC formula with a free propositional variable x. In every model M we can consider a function

$$\lambda S.\ M_{S/x}(\phi)\ :\ \mathcal{P}(D_M) \longrightarrow \mathcal{P}(D_M)$$

where $M_{S/x}$ is a model which agrees with M everywhere except x: x is interpreted as S in $M_{S/x}$. It maps each $S \subseteq D_M$ to $M_{S/p}(\phi) \subseteq D_M$.

Let us illustrate this new approach to the μ-Calculus semantics with a game example. Let ϕ be the formula $\neg fail \wedge \langle move_A\rangle(\neg fail \wedge [move_B](fail \vee win))$. Let (P, M_A, M_B, F) be a finite game of two players and $M = G_{(P,M_A,M_B,F)}$ be the corresponding model.

Let $S_0 = \emptyset$ and let S_i $(i \geq 1)$ be a set of all positions where the player A has a winning strategy against B with at most i rounds. Due to the proposition 1, $S_i = G_{(P,M_A,M_B,F)}(WIN_i)$ and $WIN_{i+1} \equiv \neg fail \wedge \langle move_A\rangle(\neg fail \wedge [move_B](fail \vee WIN_i))$ for every $i \geq 1$. It implies $M_{S_i/win}(\phi) = S_{i+1}$ for every $i \geq 0$. For every $i \geq 1$ the natural inclusion $S_i \subseteq S_{i+1}$ holds, since a i-round winning strategy is automatically a $(i+1)$-round winning strategy. Let us summarize it all as follows:

$$\begin{array}{lccccccccc}
\text{argument } S: & \emptyset & \subseteq & S_1 & \subseteq & S_2 & \subseteq & \ldots & S_i & \subseteq \ \ldots \\
\lambda S.(M_{S/win}(\phi)): & \downarrow & & \downarrow & & \downarrow & & & \downarrow & \\
\text{result } M_{S/win}(\phi): & S_1 & \subseteq & S_2 & \subseteq & S_3 & \subseteq & \ldots & S_{i+1} & \subseteq \ \ldots
\end{array}$$

As follows from the table, the mapping $\lambda S.(M_{S/win}(\phi))$ non-decreases monotonically on $\{S_i : i \geq 0\}$.

This mapping $\lambda S.(M_{S/win}(\phi))$ has another important *fixpoint* property: if $S = \bigcup_{i\geq 0} S_i$ then S is a *fixed point* of $M(\phi)$, i.e. $M_{S/win}(\phi) = S$. Informally speaking, the above equality is very natural: if the player A is in a position where he/she has a winning strategy, then he/she has a move prior to and after which the game is not lost, but after which every move of another player B leads to a position where the game is lost for him/her, or A has a winning strategy.

Are the above monotonicity and fixpoint accidental properties of special formulae in special models? Not at all! Monotonicity is a basic property of the μ-Calculus:

Proposition 5 *For any model M, sets of states $S' \subseteq S''$, propositional variable p and formula ϕ*

- *if p has no negative instances in ϕ then $M_{S'/p}(\phi) \subseteq M_{S''/p}(\phi)$,*
- *if p has no positive instances in ϕ then $M_{S''/p}(\phi) \subseteq M_{S'/p}(\phi)$.*

This property has very important semantical implications. In particular, it leads to a fixpoint characterization of semantics of μ and ν:

Proposition 6 *For any propositional variable p, any μC formula ϕ without negative instances of p, and any model $M = (D_M, (R_M, P_M))$, $M(\mu p.\phi)$ and $M(\nu p.\phi)$ are the least and the greatest fixpoints (with respect to subset inclusion $\subseteq$) of the function $(\lambda S \subseteq D_M.\ M_{S/p}(\phi)) \ : \mathcal{P}(D_M) \longrightarrow \mathcal{P}(D_M)$, which maps each $S \subseteq D_M$ to $M_{S/p}(\phi) \subseteq D_M$.*

5 Algorithmic problems for the μ-Calculus

5.1 Model checking

Let us return to the Millennium Game Puzzle. In this puzzle we are interested in a set of positions where a winning strategy exists, i.e., in states of the model $G_{2000/2001}$ where the formula WIN holds. It is a typical model checking problem, but this time for the μ-Calculus.

Let us first recall the parameters used for measuring model checking complexity and then formulate a statement about complexity. If $M = (D_M, (R_M, P_M))$ is a finite model then d_M and r_M are the number of states in D_M and edges in R_M, respectively; m is the overall model size, $(d_M + r_M)$. If ϕ is a formula, then f_ϕ is the formula size. In addition, let n_ϕ be μ and ν nesting depth of a formula ϕ.

We skip the subscripts with d, r, m, f, and n
whenever a model and/or a formula are implicit.
In contrast to EPDL,
the semantics of the μ-Calculus defined in the section 4.3
is not
a model checking algorithm for the μ-Calculus in finite models,
due to non-constructive semantics of μ and ν.
But thanks to the monotonicity property 5,
we can revise the semantics of μ and ν in finite models as follows:

7$'$. for every formula ϕ,
$M(\mu p.\ \phi) \ = \ \bigcup_{0 \le i \le d_M} M(\phi^i_p(false))$,

8′. for every formula ϕ,

$$M(\nu p.\ \phi) \ = \ \bigcap_{0\le i\le d_M} M(\phi^i_p(true)).$$

in every finite model M.
These arguments imply

Proposition 7 *The model checking problem for the μ-Calculus in finite models is decidable with an upper time bound* $O(m \times f \times d^n)$.

In particular, a computer-aided solution of the Millennium Game Puzzle, becomes just technical:
implement the above model checking
algorithm for the μ-Calculus, code the model $G_{2000/2001}$ and then "plug and play", i.e., model check the formula WIN.
The only problem is model size, but abstraction can help us again.
In accordance with the revised semantics above, every μC formula ϕ in every finite model M is equivalent to some EPDL formula ψ (which just unfolds d_M times all fixpoints in ϕ).
Hence, we have:

Proposition 8 *For all finite models M_1 and M_2, where M_1 is an abstraction of M_2 with respect to μC formulae (written within propositional and action variables Prp and Act) iff M_1 is an abstraction of M_2 with respect to EPDL formulae (written within the same propositional and action variables).*

Combined with Assertion 2,
this leads to another more efficient computer-aided solution
of the Millennium Game Puzzle: just model-check formula WIN in the model G_{2000}.

A large model size is not the single critical factor in the upper bound of the model checking complexity in the above Proposition 7. Another critical factor is the exponent, where the power depends on the input formula. A problem how to close a complexity gap between linear model checking EPDL (Proposition 2) and exponential model checking the μ-Calculus (Proposition 7) in finite models is an important research topic. Unfortunately, the best known model checking algorithms for the μ-Calculus and finite models are exponential. For example, a time bound of the Faster Model Checking Algorithm (FMC-algorithm) [7] is roughly

$$O(m \times f) \times \Big(\frac{m \times f}{a}\Big)^{a-1}$$

where an alternating depth a of a formula is the maximal number of alternating nesting μ and ν with respect to the syntactical dependences. (A formal definition is out of scope of the paper due to space limitations. We would like to point out only that the alternating depth is always less then or equal to the nesting depth for every formula: $a_\phi \leq n_\phi$.)

The best known complexity class for the model checking problem for the μ-Calculus in finite models is $\mathcal{NP} \bigcap co\text{-}\mathcal{NP}$ [10], i.e., the problem is not more complicated then satisfiability and provability in the classical propositional logic. For this reason it seems very hard to prove an exponential lower bound for the model checking problem for the μ-Calculus in finite models. Since it is not known whether the problem is $\mathcal{NP}$-complete, it seems more realistic to try to find a polynomial model checking algorithm for the μ-Calculus in finite models. At least several expressive fragments of the μ-Calculus with polynomial model checking algorithms for finite models have been identified [10, 1]. As follows from the upper bound for the FMC-algorithm, formulae with a bounded alternating nesting depth form a fragment of this kind too.

Problem 3 *(a) Describe new fragments of the μ-Calculus with a polynomial model checking in finite models. (b) Prove a polynomial upper or an exponential lower time bound for model checking the μ-Calculus in finite models.*

5.2 Decidability and Axiomatizations

Decidability is another important algorithmic problem. Its essence is to check whether a given formula of the μ-Calculus is a tautology, i.e., it is valid in *all* models. It is known that it is possible to check validity not in *all* models but in *all finite* models only due to a so-called *finite-model property* of the μ-Calculus formulae: a formula is satisfiable in a model iff it is satisfiable in a *finite* model [11]. But this reduction does not make the problem trivial! Moreover, the reduction itself is just a corollary of the decidability of the μ-Calculus with an exponential upper bound. In principle, an exponential decidability result for this logic can be proved indirectly by means of an automata-theoretic technique [29, 11]. Basically, the automata-theoretic approach comprises two stages: first, a reduction of the decidability problem for a particular logic to the emptiness problem for a particular class of automata on infinite trees, and then application of a direct decision procedure for this emptiness problem. This and other impressive applications of the automata-theoretic technique have led the program logic community to the opinion [30] that the automata-theoretic approach is the unique paradigm for proving decidability for complicated propositional program logics. In spite of the theoretical importance of the automata approach, it seems to be inefficient for implementations due to an indirect, round-about character.

Problem 4 *An efficient direct decision procedure for the μ-Calculus.*

Another complicated algorithmic problem for μ-Calculus is axiomatization. In this context we would like to remark that in the original paper [17] a natural sound axiomatization for the μ-Calculus was proposed, but the completeness of the axiomatization was proved for a fragment of this logic only. The completeness problem for the μ-Calculus was open for 10 years. Finally it was solved in [30, 31, 32] using the theory of infinite games and theory of automata on infinite trees. Nevertheless the completeness proof is very complicated and any simplifying suggestions are welcome!

Problem 5 *A complete axiomatization of the μ-Calculus made easy.*

6 Program Logics in general

6.1 What are "Program Logics"?

The logics we have discussed so far are the Elementary Propositional Dynamic Logic and the μ-Calculus. An experienced mathematician can remark that EPDL is just a polymodal variant of the classical and basic modal logic **K** [3]

and the μ-Calculus is just a polymodal variant of the $\mu\mathbf{K}$, i.e. **K** extended by fixpoints. Actually, in terms of EPDL, **K** is a variant of EPDL with a unique action variable. Since in this case a name of this variable is not important, it is possible to omit the variable in formulae and write $\Box$ and $\Diamond$ instead of [...] and $\langle ... \rangle$ respectively. These "new" modalities are read "*box*" or "*always*" and

"*diamond*" or "*sometimes*". In particular, the formulae win_i $(i \geq 0)$ for positions in the millennium game where Alice has a i-round (at most) winning strategy, are all formulae of **K** and can be rewritten in the $\Box$ and $\Diamond$ notation as $win_0 \equiv false$ and $win_{i+1} \equiv \neg fail \wedge \Diamond(\neg fail \wedge \Box(fail \vee win_i))$ for every $i \geq 1$. In this notation the following formula $\mu\ win.\ \big(\neg fail \wedge \Diamond(\neg fail \wedge \Box(fail \vee win))\big)$ of $\mu\mathbf{K}$ characterizes the set of all game positions where Alice has a winning strategy. So it is reasonable to consider EPDL as a polymodal variant the modal logic **K** and the μ-Calculus as a polymodal variant of the $\mu\mathbf{K}$. Why do people call them *program logics*? And why do we give non-mathematical names for them?

The answers are quite simple. *Program logics* are modal logics used in software and hardware verification and specification for reasoning about *programs.* In 1980s program logics comprised

- *dynamic logics* [15, 18, 16],
- *temporal logics* [27, 9],

and their extensions by means of fixpoints. EPDL is the simplest dynamic logic; μC is a very expressive extension of EPDL by fixpoints μ and ν. Temporal logics are fragments of μC with a single action variable *next* for discrete next-time. A more recent addition to the family of program logics is the *logic of knowledge* [12]. The utility of this logic is in providing a language which formalizes notions that are used informally in reasoning about multi-agent systems when a pure dynamic/temporal approach is not very convenient. The "given names" of program logics are sometimes traditional and closely related to their mathematical names[11], sometimes they are invented with respect to intuition about application domain[12]. The situation with "given names" is quite similar to the situation with a generic name of models for program logics: some researchers prefer the mathematical name *Kripke structures* while other prefer the application-oriented name *labeled transition systems.*

6.2 Why should we know program logics?

The role of the logical approach in the development of computer hardware and software increases as systems become more complex and require more effort for their specification and verification. A logical approach to verification and specification comprises the following choices:

- a specification language for property presentation,
- a formal proving technique for specified properties.

Specification languages which are in use for the presentation of properties range from propositional to high-order logics while a proving technique is either model checking (a semantical approach) or deductive reasoning (a syntactical approach).

It is possible (in principle) to construct a complete first-order axiomatization for every finite model and then try to prove a desired property (semi)automatically by means of any available logical framework [2, 21, 8]. But this purely deductive approach is sometimes not practical for complexity reasons. Let us

[11]Ex., *temporal* is a program logic, while *tense* is a basic one.

[12]Ex., *dynamic* is a program logic, while **K** is a basic one.

consider a finite model of a moderate size with approximately $100,000$ states. If it has a “clear” structure then it is reasonable to try to “capture” the model by means of a sound axiomatization and then to try to prove a desired property in a (semi)automatic style. But if a model has a “vague” structure which can be generated automatically (e.g. all possible configurations of a “small” distributed system) then it is reasonable to apply an automatic model checker to the generated system and a desired property, presented as a formula of a propositional program logic. In this case decidability and complexity issues of model checking for a particular logic arise. The problem of choosing an efficient model checking algorithm and an implementation problem follow. Efficiency issues become more important as soon as model checking is applied to huge models with, say, 10^{100} states, since with large sets representation problem arises.

We would like to give some recommendations on further reading on program logics. Some books and special chapter of handbooks can be recommended for those who are interested in the theory of program logics[13]: first [12], then [15, 16, 27, 18, 9] (in any order). There are also several books which discuss the pragmatics and applications of program logics. A comprehensive survey (from the implementation perspective) on automatic model checking techniques and applications is given in [6]. The temporal logic approach to specification and to manual deductive verification of reactive and concurrent systems is presented in [19, 20].

6.3 People and ideas in program logics

Program logics became a legitimate part of theoretical computer science and an essential element of information processing culture in the middle of the 1970s [15, 27, 18, 9]. A decade later, they were adopted by the formal method community as a convenient framework for specifying and reasoning about properties of a broad class of systems which can be presented or simulated by computer programs [19, 20, 12, 6]. Thus it is absolutely natural to pay a tribute to some people whose research were milestones in history of program logics.

Program logics as a special research domain were launched by V.R. Pratt in 1976 when he suggested *Dynamic Logic* (basically, First-Order DL) [23]. He realized that Hoare-Dijkstra *weakest pre-conditions* are modalities and incorporated weakest pre-conditions into the first-order logic as follows: for a program α and a post-condition ϕ let $([\alpha]\phi)$ be a formula which is valid in a state iff every computation of α starting in this state either diverges or terminates in another state where ϕ holds. Thus we can celebrate the 25^{th} anniversary of program logics this year.

[13]Especially for those who have not a special logical background.

A decidable propositional variant of dynamic logic — the *Propositional Dynamic Logic* — was suggested by M.J. Fisher and R.E. Ladner in 1977 [13]. A couple of years later K. Segerberg developed a sound and complete axiomatization for this logic.

A. Pnueli was the first to propose the use of temporal logic for reasoning about programs[14] [22]. His approach to the specification of concurrent and reactive systems is now well developed [19] as well as a manual deductive methodology for proving special properties [20]. This approach consists in proving properties of a program from a set of axioms that describe the behavior of the individual statements and problem-oriented inductive proof principles. Since it is a deductive approach where proofs are constructed by hand, the technique is often difficult to automate and use in practice.

Part of the reason for the further success of program logics is automatic model checking of specifications expressed in propositional level temporal logics for finite state systems [6]. Branching temporal logic CTL and polynomial model checking algorithms were developed as a new mathematical background for a new verification methodology for finite state systems by E.M. Clarke and E.A. Emerson [5], J.-P. Queille and J. Sifacis [24] in the early 1980s. An improved model checking algorithm for CTL was implemented in the EMC model checker which was able to treat models with up to 100,000 states.

At the end of the 1980s model checking researchers, encouraged by polynomial complexity of model checking for CTL in finite models and the success of model checking verification experiments for systems of a moderate size, had moved to further research topics, such as model checking for more expressive program logics (like the μ-calculus) in huge finite (10^{20} states and far beyond) and infinite models. Ordered Binary Decision Diagrams (OBDD) were created in 1987-92 [4] for handling huge finite models. OBDDs provide a canonical form for boolean formulas that is often more compact then conjunctive or disjunctive normal form and very efficient dynamic algorithms have been developed and implemented for manipulating them. A very popular modern model checker, SMV, was implemented by combining a CTL model checking algorithm with the symbolic representation of finite models. The most recent versions of SMV for UNIX, Linux, and Windows are available for download [34].

The propositional μ-Calculus was suggested by D. Kozen in 1983 [17] as a logic which can combine and unify propositional dynamic and temporal logics due to its expressive power. As was mentioned before, several complete axiomatization were developed by I. Walukiewicz in 1990s [30, 31, 32] on the basis of the theory of infinite games and the theory of automata on infinite trees.

[14]Logics of time, tense and temporal logics had been studied before A. Pnueli by philosophers and logicians, but not computer scientists.

A complete survey of program logics is outside of the scope of this paper. Only some propositional program logics were discussed, while first-order program logics were mentioned only in a brief historic survey. Due to space limitations, there is no room for more details on program logics theory, utility, and history. Nevertheless we would like to list more people, who have contributed to the theory and methodology of program logics[15]: D.Harel, J.Halpern, L.Lamport, Z.Manna, R.Parikh, J. Tiuryn, M.Vardi, etc. We would like to also remark that the most recent achievement in the theory of program logics is a sound and complete axiomatization for full branching time temporal logic CTL* by M Reynolds [25].

7 Back to the metaprogram problem

7.1 Concrete model

Now we are ready to return to the metaprogram problem and solve it with aid of the propositional μ-Calculus, model checking and abstraction.. First we would like to discuss a *concrete model GAME(N,M)* ($N \geq 1$, $M \geq 0$). Positions in this parameterized game are tuples (U, L, H, V, Q) where

- U is a set of coin numbers in $[(M+1)..(M+N)]$ which *were not* tested against other coins;
- L is a set of coin numbers in $[(M+1)..(M+N)]$ which *were* tested against other coins and were *lighter* than other coins;
- H is a set of coin numbers in $[(M+1)..(M+N)]$ which *were* tested against other coins and were *heavier* than other coins;
- V is a set of coin numbers in $[1..(N+M)]$ which are known to be *valid*;
- Q is a *balancing query*, i.e. a pair of disjoint subsets of $[1..(N+M)]$ of equal cardinality.

Three constraints are natural:

1. U, L, H and V are disjoint,
2. $U \cup L \cup H \cup V \ = \ [1..(N+M)]$,
3. $U \ \cup L \ \cup H \ \neq \ \emptyset$.

[15]it is not a complete list and, of course, it represents a personal viewpoint

In addition we can claim that

4. $U \neq \emptyset$ iff $L \cup H \ = \ \emptyset$,

5. if $Q = (S_1, S_2)$ then either $S_1 \cap V = \emptyset$ or $S_2 \cap V = \emptyset$

since (4) the single false coin is among untested coins iff all previous balancings gave equal weights, and since (5) it is not reasonable to add extra valid coins on both pans of a balance. A possible move of the player *prog* is a query for balancing two sets of coins, i.e. a pair of positions

$$\begin{tikzcd}(U,\ L,\ H,\ V,\ (\emptyset,\emptyset)) \arrow[d, "move_{prog}"] \\ (U,\ L,\ H,\ V,\ (S_1,S_2))\end{tikzcd}$$

where S_1 and S_2 are disjoint subsets of $[1..(N+M)]$ with equal cardinalities. A possible move of the player *user* is a reply $<$, $=$ or $>$ for a query which causes position change

$$\begin{tikzcd}(U,\ L,\ H,\ V,\ (S_1,S_2)) \arrow[d, "move_{user}"] \\ (U',\ L',\ H',\ V',\ (\emptyset,\emptyset)).\end{tikzcd}$$

in accordance with the query and the reply: if $S_1 \ = \ U_1 \cup L_1 \cup H_1 \cup V_1$ and $S_2 \ = \ U_2 \cup L_2 \cup H_2 \cup V_2$ respectively with $U_1, U_2 \subseteq U$, $L_1, L_2 \subseteq L$, $H_1, H_2 \subseteq H$, $V_1, V_2 \subseteq V$, then

$$U' \ = \ \begin{cases} \emptyset \text{ if the reply is } < , \\ (U \setminus (U_1 \cup U_2)) \text{ if the reply is } = , \\ \emptyset \text{ if the reply is } > , \end{cases}$$

$$L' \ = \ \begin{cases} (L_1 \cup U_1) \text{ if the reply is } < , \\ (L \setminus (L_1 \cup L_2)) \text{ if the reply is } = , \\ (L_2 \cup U_2) \text{ if the reply is } > , \end{cases}$$

$$H' \ = \ \begin{cases} (H_2 \cup U_2) \text{ if the reply is } < , \\ (H \setminus (H_1 \cup H_2)) \text{ if the reply is } = , \\ (H_1 \cup U_1) \text{ if the reply is } > , \end{cases}$$

$$V' \ = \ [1..(N+M)] \ \setminus (U' \cup L' \cup H').$$

A final position is a position $(U,\ L,\ H,\ V,\ (\emptyset,\emptyset))$ where $|U|\ +|L|\ +|H|\ =1$.

We suppose that positions, moves of the player *prog*, and final positions are modeled in the obvious way and additional comments are not required while some auxiliary intuition on moves of the player *user* is essential. Since $U \neq \emptyset$ iff $L{\cup}H\ =\ \emptyset$, there are two disjoint cases: $U\ =\ \emptyset$ XOR $L{\cup}H\ =\ \emptyset$. Let us consider the first one only, since the second is similar. In this case $U_1\ =\ U_2\ =\ U'\ =\ \emptyset$. Then

$$L'\ =\ \begin{cases} L_1 \text{ if the reply is } <\text{, since in this case a false coin is} \\ \quad\text{either in } L_1 \text{ and is lighter or it is in } H_2 \text{ and is heavier;} \\ (L \setminus (L_1 \cup L_2)) \text{ if the reply is } =\text{, since in this case a false coin is} \\ \quad\text{neither in } L_1 \text{ or } L_2 \text{ nor is it in } H_1 \text{ or } H_2\text{;} \\ L_2 \text{ if the reply is } >\text{, since in this case a false coin is} \\ \quad\text{either in } L_2 \text{ and is lighter or it is in } H_1 \text{ and is heavier;} \end{cases}$$

$$H'\ =\ \begin{cases} H_2 \text{ if the reply is } <\text{, since in this case a false coin is} \\ \quad\text{either in } L_1 \text{ and is lighter or it is in } H_2 \text{ and is heavier;} \\ (H \setminus (H_1 \cup H_2)) \text{ if the reply is } =\text{, since in this case a false coin is} \\ \quad\text{neither in } L_1 \text{ or } L_2 \text{ nor is it in } H_1 \text{ or } H_2\text{;} \\ H_1 \text{ if the reply is } >\text{, since in this case a false coin is} \\ \quad\text{either in } L_2 \text{ and is lighter or it is in } H_1 \text{ and is heavier.} \end{cases}$$

The above model is quite good from the mathematical viewpoint, but too large from the viewpoint of the computer scientist, since the number of possible positions and possible moves is an exponential function of N. Actually, for all possible U, L and H the number of possible queries ranges from 1 up to

$$Q(N)\ =\ \sum_{i=0}^{i=[\frac{N}{2}]} (C_i^N \times C_i^{N-i})$$

where all C_k^l are binomial coefficients. Since there are 2^N possible values for U and 3^N possible values for disjoint L and H, then the total amount of positions is

$$(2^N+3^N)\ \leq\ P(N)\ \leq\ (2^N+3^N)\times Q(N).$$

The amount of possible moves of the player *prog* is $P(N)$ too, while the amount of possible moves of the player *user* is bounded by the same number and triple $P(N)$. So the total amount of possible moves is

$$2\times P(N)\ \leq\ M(N)\ \leq 4\times P(N).$$

In general, the overall model size (i.e., the number of possible position and moves) of the concrete model is exponential in N. In particular, a concrete model $GAME(14,1)$ for the 15 coins puzzle is too big for explicit representation in modern personal computers.

7.2 Abstract model

Abstraction can overcome the deficit of the power of modern computers and solve the metaprogram problem: let's consider *amounts of coins* instead of *coin numbers.* This idea is natural: when somebody is solving puzzles he/she operates in terms of amounts of coins of different kinds not in terms of their numbers! Let us present this hint in formal terms as an *abstract model game(N,M)* ($N \geq 1$, $M \geq 0$). Positions in this parameterized game are tuples (u, l, h, v, q) where

- u is the amount of coins in $[1..N]$ which *were not* tested against other coins;
- l is the amount of coins in $[1..N]$ which *were* tested against other coins and turned out to be *lighter* than other coins;
- h is the amount of coins in $[1..N]$ which *were* tested against other coins and turned out to be *heavier* than other coins;
- v is an amount of coins in $[1..(N+M)]$ which are currently known to be *valid*;
- q is a *balancing query,* i.e. a pair of quadruples ((u_1, l_1, h_1, v_1) , (u_2, l_2, h_2, v_2)) of numbers of $[1..(N+M)]$.

Five constraints are absolutely natural and are closely related to constraints 1–5 for the concrete model: (1)$u+l+h \leq N$, (2)$u+l+h+v = N+M$, (3)$u+l+h \geq 1$, (4)$u \neq 0$ iff $l+h = 0$, and (5)$v_1 = 0$ or $v_2 = 0$. Additional constraints should be imposed for queries (since we can borrow coins for weighing from available untested, lighter, heavier and valid ones): (6)$u_1+u_2 \leq u$, (7)$l_1+l_2 \leq l$, (8)$h_1+h_2 \leq h$, (9)$v_1+v_2 \leq v$, (10)$u_1+l_1+h_1+v_1 = u_2+l_2+h_2+v_2$. A possible move of a player *prog* is a query for balancing two sets of coins, i.e. pair of positions

$$(u, l, h, v, ((0, 0, 0, 0), (0, 0, 0, 0))) \xrightarrow{prog} (u, l, h, v, ((u_1, l_1, h_1, v_1), (u_2, l_2, h_2, v_2))).$$

A possible move of a player *user* is a reply $<$, $=$ or $>$ for a query which causes position change

$$(u, l, h, v, ((u_1, l_1, h_1, v_1), (u_2, l_2, h_2, v_2))) \xrightarrow{user} (u', l', h', v', ((0, 0, 0, 0), (0, 0, 0, 0)))$$

in accordance with the query and a reply:

$$u' = \begin{cases} 0 \text{ if the reply is } <, \\ (u - (u_1 + u_2)) \text{ if the reply is } =, \\ 0 \text{ if the reply is } >, \end{cases}$$

$$l' = \begin{cases} (l_1 + u_1) \text{ if the reply is } < , \\ (l - (l_1 + l_2)) \text{ if the reply is } = , \\ (l_2 + u_2) \text{ if the reply is } > , \end{cases}$$

$$h' = \begin{cases} (h_2 + u_2) \text{ if the reply is } < , \\ (h - (h_1 + h_2)) \text{ if the reply is } = , \\ (h_1 + u_1) \text{ if the reply is } > , \end{cases}$$

$$v' = ((N + M) - (u' + l' + h')).$$

The final position is a position $(u,\ u,\ h,\ v,\ ((0,0,0,0),(0,0,0,0)))$ where $u + l + h = 1$. Thus the game and the corresponding abstract model are complete. The overall model size (i.e., the number of possible positions and moves) of the abstract model is polynomial in N and is less than $\frac{(N+1)^6}{6}$.

In order to utilize abstraction, let us consider both models $GAME(N, M)$ and $game(N, M)$ ($N \geq 1,\ M \geq 0$) and define a *counting* mapping $count : D_{GAME(N,M)} \to D_{game(N,M)}$ as follows:

$$count\ :\ (U, L, H, V, (S_1, S_2))\ \mapsto (|U|, |L|, |H|, q)$$

where q is

$$((|S_1 \cap U|, |S_1 \cap L|, |S_1 \cap H|, |S_1 \cap V|), (|S_2 \cap U|, |S_2 \cap L|, |S_2 \cap H|, |S_2 \cap V|)).$$

This counting mapping can be component-wise extended on pairs of positions.

Assertion 4 *For all $N \geq 1$ and $M \geq 0$ the counting mapping is a homomorphism of a labeled graph $GAME(N, M)$ onto another labeled graph $game(N, M)$ with the following property for every position (U, L, H, Q) in the $GAME(N, M)$:*

1. *count maps all moves of a player which begins in the position onto moves of the same player in the $game(N, M)$ which begins in $count(U, L, H, Q)$;*
2. *count maps all moves of a player which finishes in the position onto moves of the same player in the $game(N, M)$ which finishes in $count(U, L, H, Q)$.*

The following assertion is an immediate consequence of the above one.

Assertion 5 *For all $N \geq 1$ and $M \geq 0$ the $game(N, M)$ is an abstraction of the $GAME(N, M)$ with respect to formulae of EPDL written with use of the unique propositional variable $fail$ and two action variables $move_A$ and $move_B$.*

Finally we are ready to present a high-level model-checking-based design for the metaprogram:

1. (a) input numbers N and M of coins under question and of valid coins, the bound K for the number of balancing;

   (b) to model check formulae WIN^i_{win} for all $i \in [0..K]$ in the abstract model $game(N, M)$;

   (c) if WIN^K_{win} valid in an *initial* position $(N, 0, 0, M, ((0, 0, 0, 0), (0, 0, 0, 0)))$ then go to 2 else output `impossible` and halt;

2. output a program, which model checks formulae $\neg fail \wedge [move_B](fail \vee WIN^i_{win}(false))$ for $i \in [0..(K-1)]$ in the abstract model $game(N, M)$ and has K interactive rounds with its user as follows: for every i from K to 1 it outputs to the user a move from a *current position* to an *intermediate position* in the concrete model $GAME(N, M)$ such that

   $$count(intermediate\ position) \models_{game(N,M)}$$
   $$\models_{game(N,M)} \neg fail \wedge [move_B](fail \vee WIN^{i-1}_{win}(false));$$

   then it inputs the user's reply $<$, $=$ or $>$ and defines a *next position* for *count*(*intermediate position*) in accordance with the reply in the abstract model $game(N, M)$; finally the program reconstructs a next position $count^-$(*next position*) in the concrete model $GAME(N, M)$.

The correctness of the final high-level design follows from Proposition 5. Implement, plug and play!

References

[1] Berezine S.A., Shilov N.V. *An approach to effective model-checking of real-time finite-state machines in Mu-Calculus.* Lecture Notes in Computer Science, v.813, 1994, p.47-55.

[2] Boyer R.S., Moor J.S. *A Computational Logic.* Academic Press, 1979.

[3] Bull R.A., Segerberg K. *Basic Modal Logic.* Handbook of Philosophical Logic, v.II, Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.), p.1-88.

[4] Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J. *Symbolic Model Checking: 10^{20} states and beyond.* Information and Computation, v.98, n.2, 1992, p.142-170.

[5] Clarke E.M., Emerson E.A. *Design and Synthesis of synchronization skeletons using Branching Time Temporal Logic.* Lecture Notes in Computer Science, v.131, 1982, p.52-71.

[6] Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.

[7] Cleaveland R., Klain M., Steffen B. *Faster Model-Checking for Mu-Calculus.* Lecture Notes in Computer Science, v.663, 1993, p.410-422.

[8] Crow J., Owre S., Rushby J., Shankar N., Srivas M. *A tutorial introduction to PVS.* http://www.csl.sri.com/sri-csl-fm.html

[9] Emerson E.A. *Temporal and Modal Logic.* Handbook of Theoretical Computer Science, v.B, Elsevier and The MIT Press, 1990, p.995-1072.

[10] Emerson E.A., Jutla C.S., Sistla A.P. *On model-checking for fragments of Mu-Calculus.* Lecture Notes in Computer Science, v.697, 1993, p.385-396.

[11] Emerson E.A., Jutla C.S. *The Complexity of Tree Automata and Logics of Programs.* SIAM J. Comput., v.29, n1, 1999, p.132-158.

[12] Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995.

[13] Fisher M.J. Ladner R.E. *Propositional dynamic logic of regular programs.* J. Comput. System Sci., v.18, n.2, 1979, p.194- 211.

[14] Harel D. *First-Order Dynamic Logic.* Lecture Notes in Computer Science, v.68, 1979.

[15] Harel D. *Dynamic Logic.* Handbook of Philosophical Logic, v.II, Reidel Publishing Company, 1984 (1-st ed.), Kluwer Academic Publishers, 1994 (2-nd ed.), p.497-604.

[16] Harel D., Kozen D., Tiuryn J. *Dynamic Logic.* MIT press, 2000.

[17] Kozen D. *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.

[18] Kozen D., Tiuryn J. *Logics of Programs.* Handbook of Theoretical Computer Science, v.B, Elsevier and The MIT Press, 1990, p.789-840.

[19] Manna Z., Pnueli A. The temporal logic of Reactive and Concurrent Systems. Springer-Verlag, 1991.

[20] Manna Z., Pnueli A. Temporal verification of reactive systems: safety. Springer-Verlag, 1995.

[21] Paulson L.S. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[22] Pnueli A. *Temporal Logic of Programs.* Theoretical Computer Science, v.13, n.1, 1981, p.45-60.

[23] Pratt V.R. *Semantical Considerations on Floyd-Hoare Logic.* Proc. 17^{th} IEEE Symposium on Foundations of Computer Science, 1976, p.109-121.

[24] Queille J.-P., Sifakis J. Specification and Verification of Concurrent Systems in CESAR. LNCS, v.137, 1982, p.337-351

[25] Reynolds M. *An axiomatization of full computation tree logic.* To appear in Journal of Symbolic Logic, 2001 (a draft is available at URL `http://www.it.murdoch.edu.au/~mark/research/online`).

[26] Schlipf T., Buechner T., Fritz R., Helms M., Koehl J. *Formal verification made easy.* IBM Journal of Research & Development, v.41, n.4/5, 1997.

[27] Stirling C. *Modal and Temporal Logics.* Handbook of Logic in Computer Science, v.2, Clarendon Press, 1992, p.477-563.

[28] Streett R.S. Emerson E.A. *An Automata Theoretic Decision Procedure for the Propositional Mu-Calculus.* Information and Computation, v.81, n.3, 1989, p.249-264.

[29] Vardi M.Y. *Reasoning about the past with two-way automata'.* LNCS, v.1443, 1998, p.628-641.

[30] Walukiewicz I. *A Complete Deduction System for the μ - Calculus.* Doctoral Thesis, Warsaw, 1993.

[31] Walukiewicz I. *On completeness of the μ-calculus.* IEEE Computer Society Press, Proc. of 8-th Ann. IEEE Symposium on Logic in Computer Science, 1993, p.136-146.

[32] Walukiewicz I. *Completeness of Kozen's Axiomatization of the Propositional μ-Calculus.* Inform. and Comp., v. 157, n 3, 2000, 142-182.

[33] *ACM International Collegiate Programming Contest.* `http://acm.baylor.edu/acmicpc/default.htm`

[34] *Model Checking Code Available.*
`http://www.cs.cmu.edu/∼modelcheck/code.html`